A Dynamic Stabbing-Max Data Structure with Sub-Logarithmic Query Time

Yakov Nekrich*

Abstract

In this paper we describe a dynamic data structure that answers one-dimensional stabbing-max queries in optimal $O(\log n/\log\log n)$ time. Our data structure uses linear space and supports insertions and deletions in $O(\log n)$ and $O(\log n/\log\log n)$ amortized time respectively.

We also describe a $O(n(\log n/\log\log n)^{d-1})$ space data structure that answers d-dimensional stabbing-max queries in $O((\log n/\log\log n)^d)$ time. Insertions and deletions are supported in $O((\log n/\log\log n)^d \log\log n)$ and $O((\log n/\log\log n)^d)$ amortized time respectively.

1 Introduction

In the stabbing-max problem, a set of rectangles is stored in a data structure, and each rectangle s is assigned a priority $p(s)$. For a query point q, we must find the highest priority rectangle s that contains (or is stabbed by) q. In this paper we describe a dynamic data structure that answers stabbing-max queries on a set of one-dimensional rectangles (intervals) in optimal $O(\log n/\log\log n)$ time. We also show how this result can be extended to $d > 1$ dimensions.

Previous Work. The stabbing-max problem has important applications in networking and geographic information systems. Solutions to some special cases of the stabbing-max problem play a crucial role in classification and routing of Internet packets; we refer to e.g., [9, 11, 18] for a small selection of the previous work and to [10, 19] for more extensive surveys. Below we describe the previous works on the general case of the stabbing-max problem.

The semi-dynamic data structure of Yang and Widom [24] maintains a set of one-dimensional intervals in linear space; stabbing-max queries and insertions are supported in $O(\log n)$ time. Agarwal et al. [1] showed that stabbing-max queries on a set of one-dimensional intervals can be answered in $O(\log^2 n)$ time; the data structure of Agarwal et al. [1] uses linear space and supports updates in $O(\log n)$ time. The linear space data structure of Kaplan et al. [15] supports one-dimensional queries and insertions in $O(\log n)$ time, but deletions take $O(\log n \log\log n)$ time. In [15], the authors also consider the stabbing-max problem for a nested set of intervals: for any two intervals s_1, $s_2 \in S$, either $s_1 \subset s_2$ or $s_2 \subset s_1$ or $s_1 \cap s_2 = \emptyset$. Their data structure for a nested set of one-dimensional intervals uses $O(n)$ space and supports both queries and updates in $O(\log n)$ time. Thorup [22] described a linear space data structure that supports very fast queries, but needs $\log^{\omega(1)} n$ time to perform updates. His data structure supports stabbing-max queries in $O(\ell)$ time and updates in $O(n^{1/\ell})$ time for any parameter $l = o(\log n/\log\log n)$. However the problem of constructing a data

*Department of Computer Science, University of Chile. Supported in part by Millennium Institute for Cell Dynamics and Biotechnology (ICDB), Grant ICM P05-001-F, Mideplan, Chile. Email yakov.nekrich@googlemail.com.

structure with poly-logarithmic update time is not addressed in [22]. Agarwal et al. [2] described a data structure that uses linear space and supports both queries and updates in $O(\log n)$ time for an arbitrary set of one-dimensional intervals. The results presented in [15] and [2] are valid in the pointer machine model [20].

The one-dimensional data structures can be extended to $d > 1$ dimensions, so that space usage, query time, and update time increase by a factor of $O(\log^{d-1} n)$. Thus the best previously known data structure [2] for d-dimensional stabbing-max problem uses $O(n \log^{d-1} n)$ space, answers queries in $O(\log^d n)$ time, and supports updates in $O(\log^d n)$ time. Kaplan et al. [15] showed that d-dimensional stabbing-max queries can be answered in $O(\log n)$ time for any constant d in the special case of nested rectangles.

Our Result. The one-dimensional data structure described in [2] achieves optimal query time in the pointer machine model. In this paper we show that we can achieve sublogarithmic query time without increasing the update time in the word RAM model of computation. Our data structure supports deletions and insertions in $O(\log n/\log\log n)$ and $O(\log n)$ amortized time respectively. As follows from the lower bound of [3], any fully-dynamic data structure with poly-logarithmic update time needs $\Omega(\log n/\log\log n)$ time to answer a stabbing-max query[1]. Thus our data structure achieves optimal query time and space usage.

Our result can be also extended to $d > 1$ dimensions. We describe a data structure that uses $O(n(\log n/\log\log n)^{d-1})$ space and answers stabbing-max queries in $O((\log n/\log\log n)^d)$ time; insertions and deletions are supported in $O((\log n/\log\log n)^d \log\log n)$ and $O((\log n/\log\log n)^d)$ amortized time respectively. Moreover, our construction can be modified to support stabbing-sum[2] queries: we can count the number of one-dimensional intervals stabbed by a query point q in $O(\log n/\log\log n)$ time. The stabbing-sum data structure also supports insertions and deletions in $O(\log n)$ and $O(\log n/\log\log n)$ amortized time.

Overview. We start by describing a simple one-dimensional stabbing-max data structure in section 2. This data structure achieves the desired query and update times but needs $\omega(n^2)$ space. All intervals are stored in nodes of the base tree of height $O(\log n/\log\log n)$; the base tree is organized as a variant of the segment tree data structure. Intervals in a node u are stored in a variant of the van Emde Boas (VEB) data structure [8]. We can answer a stabbing-max query by traversing a leaf-to-root path in the base tree; the procedure spends $O(1)$ time in each node on the path.

In section 3, we show how all secondary data structures in the nodes of the base tree can be stored in $O(n \log n)$ bits of space. The main idea of our method is the compact representation of intervals stored in each node. Similar compact representations were also used in data structures for range reporting queries [6, 16] and some other problems [5]. However the previous methods are too slow for our goal: we need $O(\log\log n)$ time to obtain the representation of an element e in a node u if the representation of e in a child of u is known. Therefore it would take $O(\log n)$ time to traverse a leaf-to-root path in the base tree. In this paper we present a new, improved compact storage scheme. Using our representation, we can traverse a path in the base tree and spend $O(1)$ time in each node. We believe that our method is of independent interest and can be also applied to other problems.

[1] In fact, the lower bound of [3] is valid even for existential stabbing queries: Is there an interval in the set S that is stabbed by a query point q?

[2] The stabbing-sum problem considered in this paper is to be distinguished from the more general stabbing-group problem, in which every interval is associated with a weight drawn from a group G.

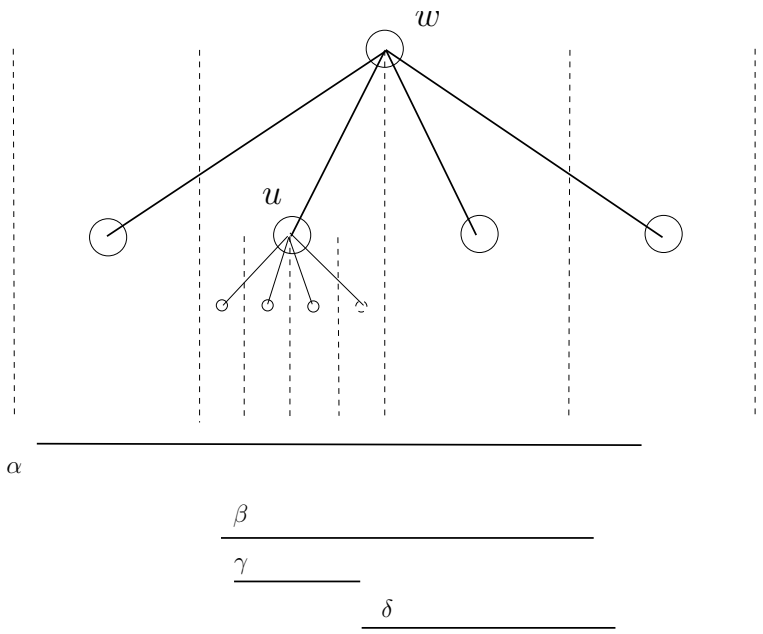

Figure 1: Internal nodes of the base tree. Intervals α and δ are stored in the set $S(w)$, but are not stored in the set $S(u)$. Interval γ is stored in $S(u)$, but γ is not stored in $S(w)$. Interval β is stored in both $S(u)$ and $S(w)$. Moreover, α, β, and δ belong to the sets $S_{23}(w)$, $S_{33}(w)$, and $S_{33}(w)$ respectively. Intervals β and γ belong to $S_{24}(u)$ and $S_{23}(u)$ respectively.

The results for multi-dimensional stabbing-max queries and stabbing-sum queries are described in Theorems 2 and 3; an extensive description of these results is provided in Appendices C and D.

2 A Data Structure with Optimal Query Time

Base Tree. In this section we describe a data structure that answers stabbing-max queries in optimal time. Endpoints of all intervals from the set S are stored in the leaves of the base tree $\mathcal{T}$. Every leaf of $\mathcal{T}$ contains $\Theta(\log^{\varepsilon} n)$ endpoints. Every internal node, except of the root, has $\Theta(\log^{\varepsilon} n)$ children; the root has $O(\log^{\varepsilon} n)$ children. Throughout this paper ε denotes an arbitrarily small positive constant. The range $rng(u)$ of a node u is an interval bounded by the minimal and maximal values stored in the leaf descendants of u.

For a leaf node u_l, the set $S(u_l)$ contains all intervals s, such that at least one endpoint of s belongs to u_l. For an internal node u, the set $S(u)$ contains all intervals s, such that $rng(u_i) \subset s$ for at least one child u_i of u but $rng(u) \not\subset s$. See Fig. 1 for an example. Thus each interval is stored in $O(\log n / \log\log n)$ sets $S(u)$. For every pair $i \leq j$, $S_{ij}(u)$ denotes the set of all intervals $s \in S(u)$ such that $rng(u_f) \subset s$ for a child u_f of u if and only if $i \leq f \leq j$. For simplicity, we will sometimes not distinguish between intervals and their priorities.

Secondary Data Structures. For each internal node u, we store a data structure $D(u)$ described in the following Proposition.

Proposition 1 *Suppose that priorities of all intervals in $S(u)$ are integers in the interval $[1, p_{\max}]$ for $p_{\max} = O(n)$. There exists a data structure $D(u)$ that uses $O(p_{\max} \cdot \log^{2\varepsilon} n)$ words of space*

and supports the following queries: for any $l \leq r$, the predecessor of q in $S_{lr}(u)$ can be found in $O(\log\log n)$ time and the maximum element in $S_{lr}(u)$ can be found in $O(1)$ time. The data structure supports insertions in $O(\log\log n)$ time. If a pointer to an interval $s \in S(u)$ is given, s can be deleted in $O(1)$ amortized time.

Proof: It suffices to store all intervals from $S_{lr}(u)$ in a VEB data structure $D_{lr}(u)$. Each $D_{lr}(u)$ uses $O(p_{\max})$ words of space and answers queries in $O(\log\log p_{\max}) = O(\log\log n)$ time [8]. It is a folklore observation that we can modify the VEB data structure so that maximum queries are supported in constant time. □

We also store a data structure $M(u)$ that contains the interval $\max_{ij}(u)$ with maximal priority among all intervals in $S_{ij}(u)$ for each $i \leq j$. Since each internal node has $\Theta(\log^{\varepsilon} n)$ children, $M(u)$ contains $O(\log^{2\varepsilon} n)$ elements. For any query index f, $M(u)$ reports the largest element among all $\max_{ij}$, $i \leq f \leq j$. In other words for any child u_f of u, $M(u)$ can find the interval with the highest priority that covers the range of the f-th child of u. Using standard techniques, we can implement $M(u)$ so that queries and updates are supported in $O(1)$ time. For completeness, we describe the data structure $M(u)$ in Appendix A.

Queries and Updates. Let π denote the search path for the query point q in the base tree $\mathcal{T}$. The path π consists of the nodes $v_0, v_1, \ldots, v_R$ where v_0 is a leaf node and v_R is the root node. Let $s(v_i)$ be the interval with the highest priority among all intervals that are stored in $\cup_{j \leq i} S(v_j)$ and are stabbed by q. The interval $s(v_0)$ can be found by examining all $O(\log^{\varepsilon} n)$ intervals stored in $S(v_0)$. Suppose that we reached a node v_i and the interval $s(v_{i-1})$ is already known. If q stabs an interval s stored in $S(v_i)$, then $rng(v_{i-1}) \subset s$. Therefore q stabs an interval $s \in S(v_i)$ if and only if s is stored in some set $S_{lr}(v_i)$ such that $l \leq f \leq r$ and v_{i-1} is the f-th child of v_i. Using the data structure $M(v_i)$, we can find in constant time the maximal interval s_m, such that $s_m \in S_{lr}(v_i)$ and $l \leq f \leq r$. Then we just set $s(v_i) = \max(s_m, s(v_{i-1}))$ and proceed in the next node v_{i+1}. The total query time is $O(\log n / \log\log n)$.

When we insert an interval s, we identify $O(\log n / \log\log n)$ nodes v_i such that s is to be inserted into $S(v_i)$. For every such v_i, we proceed as follows. We identify l and r such that s belongs to $S_{lr}(v_i)$. Using $D(v_i)$, we find the position of s in $S_{lr}(v_i)$, and insert s into $S_{lr}(v_i)$. If s is the maximal interval in $S_{lr}(v_i)$, we delete the old interval $\max_{lr}(v_i)$ from the data structure $M(v_i)$, set $\max_{lr}(v_i) = s$, and insert the new $\max_{lr}(v_i)$ into $M(v_i)$.

When an interval s is deleted, we also identify nodes v_i, such that $s \in S(v_i)$. For each v_i, we find the indices l and r, such that $s \in S_{lr}(v_i)$. Using the procedure that will be described in the next section, we can find the position of s in $S_{lr}(v_i)$. Then, s is deleted from the data structure $D(v_i)$. If $s = \max_{lr}(v_i)$, we remove $\max_{lr}(v_i)$ from $M(v_i)$, find the maximum priority interval in $S_{lr}(v_i)$, and insert it into $M(v_i)$. We will show in section 3 that positions of the deleted interval s in $S_{lr}(v_i)$ for all nodes v_i can be found in $O(\log n / \log\log n)$ time. Since all other operations take $O(1)$ time per node, the total time necessary for a deletion of an interval is $O(\log n / \log\log n)$.

Unfortunately, the space usage of the data structure described in this section is very high: every VEB data structure $D_{lr}(u)$ needs $O(p_{\max})$ space, where $p_{\max}$ is the highest possible interval priority. Even if $p_{\max} = O(n)$, all data structures $D(u)$ use $O(n^2 \log^{2\varepsilon} n)$ space. In the next section we show how all data structures $D(u)$, $u \in \mathcal{T}$, can be stored in $O(n \log n)$ bits without increasing the query and update times.

3 Compact Representation

The key idea of our compact representation is to store only interval identifiers in every node u of $\mathcal{T}$. Our storage scheme enables us to spend $O(\log\log n)$ bits for each identifier stored in a node u. Using the position of an interval s in a node u, we can obtain the position of s in the parent w of u. We can also compare priorities of two intervals stored in the same node by comparing their positions. These properties of our storage scheme enable us to traverse the search path for a point q and answer the query as described in section 2.

Similar representations were also used in space-efficient data structures for orthogonal range reporting [6, 16] and orthogonal point location and line-segment intersection problems [5]. Storage schemes of [16, 5] also use $O(\log\log n)$ bits for each interval stored in a node of the base tree. The main drawback of those methods is that we need $O(\log\log n)$ time to navigate between a node and its parent. Therefore, $\Theta(\log n)$ time is necessary to traverse a leaf-to-root path and we need $\Omega(\log n)$ time to answer a query. In this section we describe a new method that enables us to navigate between nodes of the base tree and update the lists of identifiers in $O(1)$ time per node. The main idea of our improvement is to maintain identifiers for a set $\overline{S}(u) \supset S(u)$ in every $u \in \mathcal{T}$. When an interval is inserted in $S(u)$, we also add its identifier to $\overline{S}(u)$. But when an interval is deleted from $S(u)$, its identifier is not removed from $\overline{S}(u)$. When the number of deleted interval identifiers in all $\overline{S}(u)$ exceeds the number of intervals in $S(u)$, we re-build the base tree and all secondary structures (global re-build).

Compact Lists. We start by defining a set $S'(u) \supset S(u)$. If u is a leaf node, then $S'(u) = S(u)$. If u is an internal node, then $S'(u) = S(u) \cup (\cup_i S'(u_i))$ for all children u_i of u. An interval s belongs to $S'(u)$ if at least one endpoint of s is stored in a leaf descendant of u. Hence, $|\cup_v S'(v)| = O(n)$ where the union is taken over all nodes v that are situated on the same level of the base tree $\mathcal{T}$. Since the height of $\mathcal{T}$ is $O(\log n/\log\log n)$, the total number of intervals stored in all $S'(u)$, $u \in \mathcal{T}$, is $O(n\log n/\log\log n)$.

Let $\overline{S}(u)$ be the set that contains all intervals from $S'(u)$ that were inserted into $S'(u)$ since the last global re-build. We will organize global re-builds in such way that at most one half of elements in all $\overline{S}(u)$ correspond to deleted intervals. Therefore the total number of intervals in $\cup_{u\in T}\overline{S}(u)$ is $O(n\log n/\log\log n)$. We will show below how we can store the represntations of sets $\overline{S}(u)$ in compact form, so that an element of $\overline{S}(u)$ uses $O(\log\log n)$ bits in average. Since $S(u) \subset \overline{S}(u)$, we can also use the same method to store all $S(u)$ and $D(u)$ in $O(n\log n)$ bits.

Sets $S'(u)$ and $\overline{S}(u)$ are not stored explicitly in the data structure. Instead, we store a list $\texttt{Comp}(u)$ that contains a compact representation for identifiers of intervals in $\overline{S}(u)$. $\texttt{Comp}(u)$ is organized as follows. The set $\overline{S}(u)$ is sorted by interval priorities and divided into blocks. If $|\overline{S}(u)| > \log^3 n/2$, then each block of $\overline{S}(u)$ contains at least $\log^3 n/2$ and at most $2\log^3 n$ elements. Otherwise all $e \in \overline{S}(u)$ belong to the same block. Each block B is assigned an integer block label $\texttt{lab}(B)$ according to the method of [14, 23]. Labels of blocks are monotone with respect to order of blocks in $\texttt{Comp}(u)$: The block B_1 precedes B_2 in $\texttt{Comp}(u)$ if and only if $\texttt{lab}(B_1) < \texttt{lab}(B_2)$. Besides that, all labels assigned to blocks of $\texttt{Comp}(u)$ are bounded by a linear function of $|\texttt{Comp}(u)|/\log^3 n$: for any block B in $\texttt{Comp}(u)$, $\texttt{lab}(B) = O(|\texttt{Comp}(u)|/\log^3 n)$. When a new block is inserted into a list, we may have to change the labels of $O(\log^2 n)$ other blocks.

For every block B, we store its block label as well as the pointers to the next and the previous blocks in the list $\texttt{Comp}(u)$. For each interval $\bar{s}$ in a block B of $\overline{S}(u)$, the list $\texttt{Comp}(u)$ contains the *identifier* of $\bar{s}$ in $\texttt{Comp}(u)$. The identifier is simply the list of indices of children u_i of u, such that $\bar{s} \in \texttt{Comp}(u_i)$. As follows from the description of the base tree and the sets $\overline{S}(u)$, we store at most

two child indices for every interval $\bar{s}$ in a block; hence, each identifier uses $O(\log \log n)$ bits. To simplify the description, we sometimes will not distinguish between an interval s and its identifier in a list $\texttt{Comp}(u)$.

We say that the position of an interval s in a list $\texttt{Comp}(u)$ is known if the block B that contains s and the position of s in B are known. If we know positions of two intervals s_1 and s_2 in $\texttt{Comp}(u)$, we can compare their priorities in $O(1)$ time. Suppose that s_1 and s_2 belong to blocks B_1 and B_2 respectively. Then $s_1 > s_2$ if $\texttt{lab}(B_1) > \texttt{lab}(B_2)$, and $s_1 < s_2$ if $\texttt{lab}(B_1) < \texttt{lab}(B_2)$. If $\texttt{lab}(B_1) = \texttt{lab}(B_2)$, we can compare priorities of s_1 and s_2 by comparing their positions in the block $B_1 = B_2$.

The rest of this section has the following structure. First, we describe auxiliary data structures that enable us to search in a block and navigate between nodes of the base tree. Each block B contains a poly-logarithmic number of elements and every identifier in B uses $O(\log \log n)$ bits. We can use this fact and implement block data structures, so that queries and updates are supported in $O(1)$ time. If the position of some $\bar{s}$ in a list $\texttt{Comp}(u)$ is known, we can find in constant time the positions of $\bar{s}$ in the parent of u. Next, we show how data structures $D(u)$ and $M(u)$, defined in section 2, are modified. Finally, we describe the search and update procedures.

Block Data Structures. We store a data structure $F(B)$ that supports rank and select queries in a block B of $\texttt{Comp}(u)$: A query $\texttt{rank}(f, i)$ returns the number of intervals that also belong to $\overline{S}(u_f)$ among the first i elements of B. A query $\texttt{select}(f, i)$ returns the smallest j, such that $\texttt{rank}(f, j) = i$; in other words, $\texttt{select}(f, i)$ returns the position of the i-th interval in B that also belongs to $\overline{S}(u_f)$. We can answer $\texttt{rank}$ and $\texttt{select}$ queries in a block in $O(1)$ time. We can also count the number of elements in a block of $\texttt{Comp}(u)$ that are stored in the f-th child of u, and determine for the r-th element of a block in which children of u it is stored. Implementation of $F(B)$ will be described in Appendix A.

For each block $B_j \in \texttt{Comp}(u)$ and for any child u_f of u, we store a pointer to the largest block B_j^f before B_j that contains an element from $\texttt{Comp}(u_f)$. These pointers are maintained with help of a data structure $P_f(u)$ for each child u_f. We implement $P_f(u)$ as an incremental split-find data structure [13]. Insertion of a new block label into $P_f(u)$ takes $O(1)$ amortized time; we can also find the block B_j^f for any block $B_j \in \texttt{Comp}(u)$ in $O(1)$ worst-case time. Using the data structure $F(B_j)$ for a block B_j, we can identify for any element e in a block B_j the largest $e_f \leq e$, $e_f \in B_j$, such that e_f belongs to $\texttt{Comp}(u_f)$. Thus we can identify for any $e \in \texttt{Comp}(u)$ the largest element $e_f \in \texttt{Comp}(u)$, such that $e_f \leq e$ and $e_f \in \texttt{Comp}(u_f)$, in $O(1)$ time.

Navigation in Nodes of the Base Tree. Finally, we store pointers to selected elements in each list $\texttt{Comp}(u)$. Pointers enable us to navigate between nodes of the base tree: if the position of some $e \in \texttt{Comp}(u)$ is known, we can find the position of e in $\texttt{Comp}(w)$ for the parent w of u and the position of e in $\texttt{Comp}(u_i)$ for any child u_i of u such that $\texttt{Comp}(u_i)$ contains e.

We associate a *stamp* $t(e)$ with each element stored in a block B; every stamp is a positive integer bounded by $O(|B|)$. A pointer to an element e in a block B consists of the block label $\texttt{lab}(B)$ and the stamp of e in B. When an element is inserted into a block, stamps of other elements in this block do not change. Therefore, when a new interval is inserted into a block B we do not have to update all pointers that point into B. Furthermore, we store a data structure $H(B)$ for each block B. Using $H(B)$, we can find the position of an element e in B if its stamp $t(e)$ in B is known. If an element e must be inserted into B after an element e_p, then we can assign a stamp t_e to e and insert it into $H(B)$ in $O(1)$ time. Implementation of $H(B)$ is very similar to the implementation of $F(B)$ and will be described in the full version.

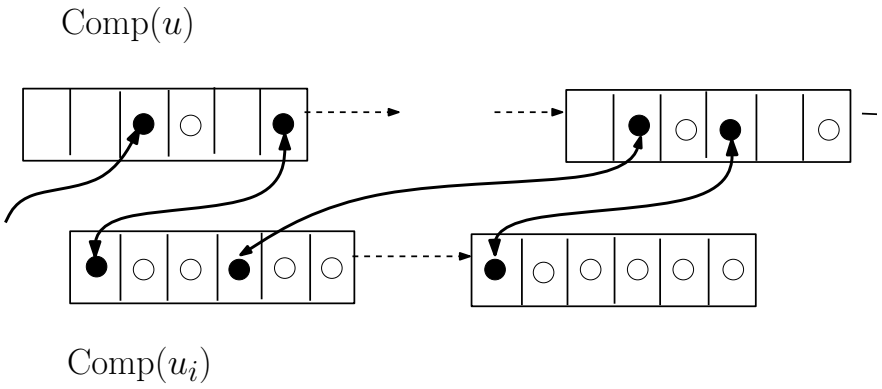

Figure 2: Parent and child pointers in blocks of $\texttt{Comp}(u)$ and $\texttt{Comp}(u_i)$. Intervals that are stored in $\texttt{Comp}(u_i)$ are depicted by circles; filled circles are intervals with pointers. Only relevant intervals and pointers are shown in $\texttt{Comp}(u)$. In $\texttt{Comp}(u_i)$, only parent pointers are shown.

If e is the first element in the block B of $\texttt{Comp}(u)$ that belongs to $\texttt{Comp}(u_i)$, then we store the pointer from the copy of e in $\texttt{Comp}(u)$ to the copy of e in $\texttt{Comp}(u_i)$. If an element $e \in \texttt{Comp}(u)$ is also stored in $\texttt{Comp}(u_i)$ and e is the first element in a block B' of $\texttt{Comp}(u_i)$, then there is a pointer from the copy of e in $\texttt{Comp}(u)$ to the copy of e in $\texttt{Comp}(u_i)$. Such pointers will be called child pointers. For any pointer from $e \in \texttt{Comp}(u)$ to $e \in \texttt{Comp}(u_i)$, we store a pointer from $e \in \texttt{Comp}(u_i)$ to $e \in \texttt{Comp}(u)$. Such pointers will be called parent pointers. See Fig. 2 for an example.

We can store each pointer in $O(\log n)$ bits. The total number of pointers in $\texttt{Comp}(u)$ equals to the number of blocks in $\texttt{Comp}(u)$ and $\texttt{Comp}(u_i)$ for all children u_i of u. Hence, all pointers and all block data structures use $O(n \log n)$ bits.

If we know the position of some interval s in $\texttt{Comp}(v)$, we can find the position of s in the parent w of v as follows. Suppose that an interval s is stored in the block B of $\texttt{Comp}(v)$ and v is the f-th child of w. We find the last interval s' in B stored before s, such that there is a parent pointer from s'. Let m denote the number of elements between s' and s in B. Let B' be the block in $\texttt{Comp}(w)$ that contains s'. Using $H(B')$, we find the position m' of s' in the block B' of $\texttt{Comp}(w)$. Then the position of s in B' can be found by answering the query $\texttt{select}(f, \texttt{rank}(f, m') + m)$ to a data structure $F(B')$. Using a symmetric procedure, we can find the position of s in $\texttt{Comp}(v)$ if its position in $\texttt{Comp}(w)$ is known.

Root and Leaves of the Base Tree. Elements of $\overline{S}(v_R)$ are explicitly stored in the root node v_R. That is, we store a table in the root node v_R that enables us to find for any interval s the block B that contains the identifier of s in $\texttt{Comp}(v_R)$ and the stamp of s in B. Conversely, if the position of s in a block B of $\texttt{Comp}(v_R)$ is known, we can find its stamp in B in $O(1)$ time. If the block label and the stamp of an interval s are known, we can identify s in $O(1)$ time. In the same way, we explicitly store the elements of $\overline{S}(u_l)$ for each leaf node u_l. Moreover, we store all elements of $\overline{S}(v_R)$ in a data structure R, so that the predecessor and the successor of any value x can be found in $O(\log n / \log \log n)$ time.

Data Structures $M(u)$ and $D(u)$. Each set $S(u)$ and data structure $D(u)$ are also stored in compact form. $D(u)$ consists of structures $D_{lr}(u)$ for every pair $l \leq r$. If a block B contains a label of an interval $s \in S_{lr}(u)$, then we store the label of B in the VEB data structure $D_{lr}(u)$. The data

structure $G(B)$ contains data about intervals in $S(u) \cap B$. For every $s \in G(B)$, we store indices l, r if $s \in S_{lr}(u)$; if an element $s \in B$ does not belong to S (i.e., s was already deleted), then we set $l = r = 0$. For a query index f, $G(B)$ returns in $O(1)$ time the highest priority interval $s \in B$, such that $s \in S_{lr}(u)$ and $l \leq f \leq r$. A data structure $G(B)$ uses $O(|B| \log\log n)$ bits of space and supports updates in $O(1)$ time. Implementation of $G(B)$ is very similar to the implementation of $F(B)$ and will be described in the full version of this paper. We store a data structure $M(u)$ in every node $u \in \mathcal{T}$, such that $\texttt{Comp}(u)$ consists of at least two blocks. For each pair $l \leq r$, the data structure $M(u)$ stores the label of the block that contains the highest priority interval in $S_{lr}(u)$. For a query f, $M(u)$ returns the label of the block that contains the highest priority interval in $\cup_{l \leq f \leq r} S_{lr}(u)$. Such queries are supported in $O(1)$ time; a more detailed description of the data structure $M(u)$ will be given in Appendix A.

Search Procedure. We can easily modify the search procedure of section 2 for the case when only lists $\texttt{Comp}(u)$ are stored in each node. Let v_i be a node on the path $\pi = v_0, \ldots v_R$, where π is the search path for the query point q. Let $s(v_i)$ denote the interval that has the highest priority among all intervals that belong to $\cup_{j \leq i} S(v_j)$ and are stabbed by q. We can examine all intervals in $S(v_0)$ and find $s(v_0)$ in $O(\log^{\varepsilon} n)$ time. The position of $s(v_0)$ in $\texttt{Comp}(v_0)$ can also be found in $O(1)$ time. During the i-th step, $i \geq 1$, we find the position of $s(v_i)$ in $\texttt{Comp}(v_i)$. An interval s in $S(v_i)$ is stabbed by q if and only if $s \in S_{lr}(v_i)$, $l \leq f \leq r$, and v_{i-1} is the f-th child of v_i. Using $M(v_i)$, we can find the block label of the maximal interval s_m among all intervals stored in $S_{lr}(v_i)$ for $l \leq f \leq r$. Let B_m be the block that contains s_m. Using the data structure $G(B_m)$, we can find the position of s_m in B_m.

By definition, $s(v_i) = \max(s_m, s(v_{i-1}))$. Although we have no access to s_m and $s(v_{i-1})$, we can compare their priorities by comparing positions of s_m and $s(v_{i-1})$ in $\texttt{Comp}(v_i)$. Since the position of $s(v_{i-1})$ in $\texttt{Comp}(v_{i-1})$ is already known, we can find its position in $\texttt{Comp}(v_i)$ in $O(1)$ time. We can also determine, whether s_m precedes or follows $s(v_{i-1})$ in $\texttt{Comp}(v_i)$ in $O(1)$ time. Since a query to $M(v_i)$ also takes $O(1)$ time, our search procedure spends constant time in each node v_i for $i \geq 1$. When we know the position of $s(v_R)$ in $\texttt{Comp}(v_R)$, we can find the interval $s(v_R)$ in $O(1)$ time. The interval $s(v_R)$ is the highest priority interval in S that is stabbed by q. Hence, a query can be answered in $O(\log n / \log\log n)$ time.

We describe how our data structure can be updated in section 4. Our result is summed up in the following Theorem.

Theorem 1 *There exists a linear space data structure that answers orthogonal stabbing-max queries in $O(\log n / \log\log n)$ time. This data structure supports insertions and deletions in $O(\log n)$ and $O(\log n / \log\log n)$ amortized time respectively.*

4 Updates in the Data Structure of Theorem 1

Suppose that an interval s is inserted into S. The insertion procedure consists of two parts. First, we insert s into lists $\texttt{Comp}(v_i)$ and update $\texttt{Comp}(v_i)$ for all relevant nodes v_i of $\mathcal{T}$. Then, we insert s into data structures $D(v_i)$ and $M(v_i)$.

Using the data structure R, we can identify the segment $s'(v_R)$ that precedes s in $\overline{S}(v_R)$. Then, we find the position of $s'(v_R)$ in $\texttt{Comp}(v_R)$. We also find the leaves v_a and v_b of $\mathcal{T}$, in which the left and the right endpoints of s must be stored.

The path $\pi = v_R, \ldots, v_a$ is traversed starting at the root node. Let $s'(v_i)$ be the segment that precedes s in $\texttt{Comp}(v_i)$ and let $B(v_i)$ be the block that contains $s'(v_i)$. In every node v_i, we insert

the identifier for s after $s'(v_i)$. We also update data structures $F(B(v_i))$ and $H(B(v_i))$ for the block $B(v_i)$ that contains s. If the number of intervals in $B(v_i)$ equals $2\log^3 n$, we split the block $B(v_i)$ into two blocks $B_1(v_i)$ and $B_2(v_i)$ of equal size. We assign a new label to $B_2(v_i)$ and update the labels for some blocks of $\texttt{Comp}(v_i)$ in $O(\log^2 n)$ time. We also may have to update $O(\log^\varepsilon n)$ split-find data structures $P_f(v_i)$. Besides that, $O(1)$ child pointers in the parent of v_i and $O(\log^\varepsilon n)$ parent pointers in the children of v_i may also be updated. We split the block after $\Theta(\log^3 n)$ insertions; hence, an amortized cost of splitting a block is $O(1)$. When s is inserted into $\texttt{Comp}(v_i)$, we find the largest element $s'(v_{i-1}) \leq s$, such that $s'(v_{i-1})$ is also stored in $\texttt{Comp}(v_{i-1})$. Then, we find the position of $s'(v_{i-1})$ in $\texttt{Comp}(v_{i-1})$ and proceed in the node v_{i-1}.

Let v_c be the lowest common ancestor of v_a and v_b, and let π_b be the path from v_c to v_b. We also insert s into $\texttt{Comp}(v_j)$ for all $v_j \in \pi_b$; nodes $v_j \in \pi_b$ are processed in the same way as nodes $v_i \in \pi_a$

During the second stage, we update data structures $D(v_i)$ and $M(v_i)$ for $v_i \in \pi_a \cup \pi_b$. For any such v_i we already know the block $B \in \texttt{Comp}(v_i)$ that contains s and the position of s in B. Hence, a data structure $G(B)$ can be updated in $O(1)$ time. If s is the only interval in B that belongs to $S_{lr}(v_i)$ for some l, r, we insert $\texttt{lab}(B)$ into $D_{lr}(v_i)$. If $\texttt{lab}(B)$ is the greatest label in $D_{lr}(v_i)$, we also update $M(v_i)$. An insertion into a data structure $D_{lr}(v_i)$, $v_i \in \pi_a \cup \pi_b$, takes $O(\log\log n)$ time. Updates of all other structures in v_i take $O(1)$ time. Hence, an insertion takes $O(\log n)$ time in total.

When an interval is deleted, we identify its position in every relevant node v_i. This can be done in the same way as during the first stage of the insertion procedure. Then, we update the data structures $D(v_i)$, $M(v_i)$, and $G(B(v_i))$ if necessary. Since a block label can be removed from $D_{lr}(v_i)$ in $O(1)$ time, the total time necessary to delete an interval is $O(\log n/\log\log n)$.

Re-Balancing of the Tree Nodes. It remains to show how the tree can be rebalanced after updates, so that the height of $\mathcal{T}$ remains $O(\log n/\log\log n)$. We implement the base tree $\mathcal{T}$ as a weight-balanced B-tree [4] with branching parameter ϕ and leaf parameter ϕ for $\phi = \log^\varepsilon n$. We assume that the constant $\varepsilon < 1/8$.

When a segment s is deleted, we simply mark it as deleted. After $n/2$ deletions, we re-build the base tree and all data structures stored in the nodes. This can be done in $O(n\log n/\log\log n)$ time.

Now we show how insertions can be handled. We denote by the weight of u the total number of elements in the leaf descendants of u. The weight n_u of u also equals to the number of segment identifiers in $\texttt{Comp}(u)$. Our weight-balanced B-tree is organized in such way that $n_u = \Theta(\phi^{\ell+1})$, where $\phi = \log^\varepsilon n$ and ℓ is the level of a node u. A node is split after $\Theta(\phi^{\ell+1})$ insertions; when a node u is split into u' and u'', the ranges of the other nodes in the base tree do not change; see [4] for a detailed description of node splitting. We will show that all relevant data structures can be re-built in $O(n_u)$ time. Hence, the amortized cost of splitting a node is $O(1)$. When a segment is inserted, it is inserted into $O(\log n/\log\log n)$ nodes of the base tree. Hence, the total amortized cost of all splitting operations caused by inserting a segment s into our data structure is $O(\log n/\log\log n)$.

It remains to show how secondary data structures are updated when a node u is split. Let w be the parent of u. The total number of segments in $S(u')$ and $S(u'')$ does not exceed $|S(u)| = O(n_u)$. Hence, we can construct data structures $D(u')$, $D(u'')$ and lists $\texttt{Comp}(u')$, $\texttt{Comp}(u'')$ with all block data structures in $O(|S(u)|) = O(n_u)$ time. We can find the positions of all elements $e \in \texttt{Comp}(u') \cup \texttt{Comp}(u'')$ in $\texttt{Comp}(w)$, update their identifiers in $\texttt{Comp}(w)$, and update all auxiliary data structures in $O(n_u)$ time.

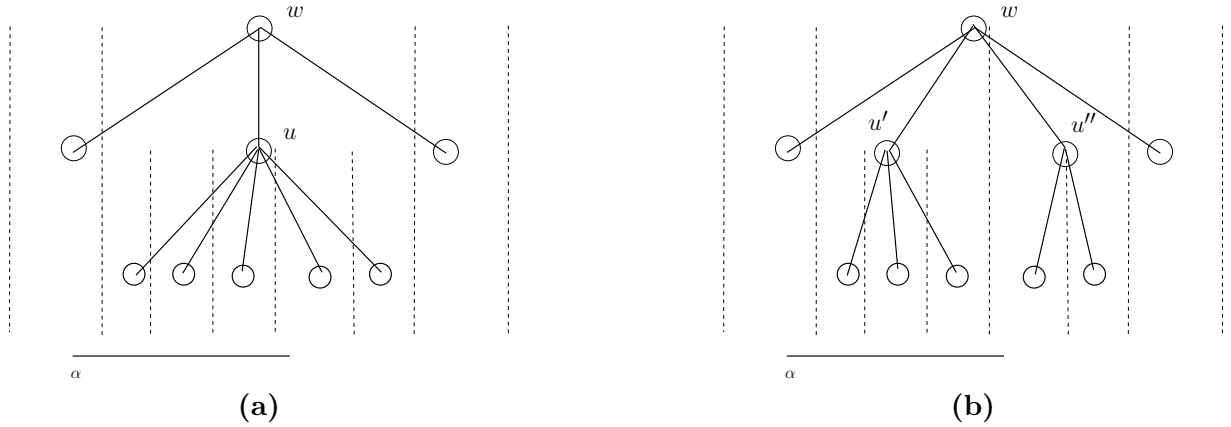

Figure 3: Node u is split into u' and u''. Segment α is moved from $S(u)$ to $S(w)$ after splitting of a node u.

Some of the intervals stored in $S(u)$ can be moved from $S(u)$ to $S(w)$: if an interval $s \in S(u)$ does not cover $rng(u)$ but covers $rng(u')$ or $rng(u'')$, then s must be stored in $S(w)$ after splitting. See Fig. 3 for an example. The total number of elements in $\texttt{Comp}(w)$ is $\Theta(\phi^{\ell+2}) = \Theta(n_u \cdot \phi)$; hence, the total number of blocks in $\texttt{Comp}(w)$ is $o(n_u/\log^3 n)$. Identifiers of all segments in $S(u)$ are already stored in $\texttt{Comp}(w)$. We can update all block data structures in $O(1)$ time per segment. Every update of the data structure $D(w)$ takes $O(\log\log n)$ time. But the total number of insertions into $D(w)$ does not exceed the number of blocks in $D(w)$. Therefore the total time needed to update $D(w)$ is $O(n_u \log^{\varepsilon-3} n) = o(n_u)$.

Suppose that u was the k-th child of w. Some segments stored in $S_{lk}(w)$ for $l < k$ can be moved to $S_{l(k+1)}(w)$, and some segments in $S_{kr}(w)$ for $r > k$ can be moved to $S_{(k+1)r}(w)$. Since the total number of blocks in $\texttt{Comp}(w)$ is $O(n_u/\log^3 n)$, all updates of data structures $D_{ij}(w)$ take $O((n_u/\log^3 n)\log\log n) = o(n_u)$ time. At most n_u segments were stored in $S_{lk}(w)$ and $S_{kr}(w)$ before the split of u. Hence, we can update all block data structures in $O(n_u)$ time.

Finally, we must change the identifiers of segments stored in $\texttt{Comp}(w)$ and data structures $G(B)$. Recall that an identifier indicates in which children of w a segment s is stored. Suppose that a segment s had an identifier $r > k$ in $\texttt{Comp}(w)$. Then its identifier will be changed to $r+1$ after the split of u. The total number of segments with incremented identifiers does not exceed $n_w = O(n_u \cdot \phi)$. However, each identifier is stored in $O(\log\log n)$ bits. We can use this fact, and increment the values of $\sqrt{\log n}$ identifiers in $O(1)$ time using the following look-up table T. There are $O((\log n)^{\sqrt{\log n}})$ different sequences of $\sqrt{\log n}$ identifiers. For every such sequence α and any $j = O(\log^{\varepsilon} n)$, $T[\alpha][j] = \alpha'$. Here α' is the sequence that we obtain if every $j' > j$ in the sequence α is replaced with $j'+1$. Thus the list $\texttt{Comp}(w)$ can be updated in $O(n_w/\sqrt{\log n}) = O(n_u)$ time. Using the same approach, we can also update data structures $F(B)$ and $G(B)$ for each block B in $O(|B|/\sqrt{\log n})$ time. Hence, the total time necessary to update all data structures in w because some identifiers must be incremented is $O(n_u)$.

Since all data structures can be updated in $O(n_u)$ time when a node u is split, the total amortized cost of an insertion is $O(\log n)$.

5 Multi-Dimensional Stabbing-Max Queries

Our data structure can be extended to the case of d-dimensional stabbing-max queries for $d > 1$. In this section we prove the following theorem.

Theorem 2 *There exists a data structure that uses $O(n(\log n/\log\log n)^{d-1})$ space and answers d-dimensional stabbing-max queries in $O((\log n/\log\log n)^d)$ time. Insertions and deletions are supported in $O((\log n/\log\log n)^d \log\log n)$ and $O((\log n/\log\log n)^d)$ amortized time respectively.*

Let (d, g)-stabbing problem denote the $(d+g)$-dimensional stabbing-max problem for the case when the first d coordinates can assume arbitrary values and the last g coordinates are bounded by $\log^\rho n$, for $\rho \leq \frac{1}{8(g+d)}$. First, we show that the data structure of Theorem 1 can be modified to support $(1, g)$-stabbing queries for any constant g. Then, we will show how arbitrary d-dimensional queries can be answered. Throughout this section $proj_i(s)$ denotes the projection of a rectangle s on the i-th coordinate. We denote by $b_i(s)$ and $e_i(s)$ the i-th coordinates of endpoints in a rectangle s, so that $proj_i(s) = [b_i(s), e_i(s)]$.

A Data Structure for the $(1, g)$-Stabbing Problem. The solution of the $(1, g)$-stabbing problem is very similar to our solution of the one-dimensional problem. We construct the base tree $\mathcal{T}$ on the x-coordinates of all rectangles. $\mathcal{T}$ is defined as in section 2, but we assume that $\varepsilon \leq \rho$. Sets $S(u)$, $S_{lr}(u)$, and $\overline{S}(u)$ are defined as in sections 2, 3. We define $S_{lr}[j_1, \ldots, j_g, \ldots, j_{2g}](u)$ as the set of rectangles s in $S_{lr}(u)$ such that $b_2(s) = j_1$, ..., $b_{g+1}(s) = j_g$, $e_2(s) = j_{g+1}$, ..., $e_{g+1}(s) = j_{2g}$.

A rectangle identifier for a rectangle s stored in a list $\texttt{Comp}(u)$ consists of one or two tuples with $2g + 1$ components. The first component of each tuple is an index j, such that s belongs to the j-th child of u; remaining $2g$ components are the last g coordinates of the rectangle endpoints. The data structure $M(u)$ contains the maximum priority rectangle in $S_{lr}[j_1, \ldots, j_{2g}](u)$ for each $l, r, j_1, \ldots, j_{2g}$. $M(u)$ can find for any $(f, h_1, \ldots, h_g)$ the highest priority rectangle in all $S_{lr}[j_1, \ldots, j_{2g}](u)$, such that $l \leq f \leq r$, and $j_i \leq h_i \leq j_{g+i}$ for $1 \leq i \leq g$. We can implement $M(u)$ as for the one-dimensional data structure. Data structure $D(u)$ consists of VEB structures $D_{lr}[j_1, \ldots, j_{2g}](u)$. If a block B of $\texttt{Comp}(u)$ contains a rectangle from $S_{lr}[j_1, \ldots, j_{2g}](u)$, then $\texttt{lab}(B)$ is stored in $D_{lr}[j_1, \ldots, j_{2g}](u)$.

We can answer a query $q = (q_x, q_1, \ldots, q_g)$ by traversing the search path $\pi = v_0, \ldots, v_R$ for q_x in the base tree $\mathcal{T}$. As in Theorem 1, we start at the leaf node v_0 and find the maximal rectangle stored in $S(v_l)$ that is stabbed by the query point q. The search procedure in nodes $v_1, \ldots, v_R$ is organized in the same way as the search procedure in section 3: The rectangle $s(v_i)$ is defined as in section 3. If v_{i-1} is the f-th child of v_i, we answer the query $(f, q_1, \ldots, q_g)$ using the data structure $M(v_i)$. Then we visit the block B_m returned by $M(v_i)$ and find the last rectangle s_m in B_m with an identifier $(f, b_2(s_m), \ldots, b_{g+1}(s_m), e_2(s_m), \ldots, e_{g+1}(s_m))$ such that $b_{i+1}(s_m) \leq q_i \leq e_{i+1}(s_m)$ for $1 \leq i \leq g$. Finally we compare s_m with the rectangle $s(v_{i-1})$ by comparing their positions in $\texttt{Comp}(v_i)$. Thus we can find the position of $s(v_i)$ in $\texttt{Comp}(v_i)$. When we reach the root v_R, $s(v_R)$ is the highest priority segment that is stabbed by q.

Updates of the list $\texttt{Comp}(u)$ and all auxiliary data structures are implemented as in section 4.

Lemma 1 *There exists a data structure that answers $(1, g)$-stabbing queries in $O(\log n/\log\log n)$ time and uses $O(n)$ space. Insertions and deletions are supported in $O(\log n)$ and $O(\log n/\log\log n)$ amortized time respectively.*

A Data Structure for the (d, g)-Stabbing Problem. The result for $(1, g)$-stabbing can be extended to (d, g)-stabbing queries using the following Lemma.

Lemma 2 *Suppose that there is a $O(n(\log n/\log\log n)^{d-2})$ space data structure D_1 that answers $(d-1,g+1)$-stabbing queries in $O((\log n/\log\log n)^{d-1})$ time; D_1 supports insertions and deletions in $O((\log n/\log\log n)^{d-1}\log\log n)$ and $O((\log n/\log\log n)^{d-1})$ amortized time respectively.*
Then there exists a $O(n(\log n/\log\log n)^{d-1})$ space data structure D_2 that answers (d,g)-stabbing queries in $O((\log n/\log\log n)^{d})$ time; D_2 supports insertions and deletions in amortized time $O((\log n/\log\log n)^{d}\log\log n)$ and $O((\log n/\log\log n)^{d})$ respectively.

The main idea is to construct the base tree $\mathcal{T}$ on the d-th coordinates of rectangles and store a data structure for $(d-1,g+1)$-stabbing queries in each tree node. The tree is organized as in section 2 and the first part of this section. Let q_d denote the d-th coordinate of a point. Leaves contain d-th coordinates of rectangle endpoints. A rectangle s is stored in a set $S(u)$ for a leaf u if $proj_d(s)\cap rng(u)\neq\emptyset$ and $rng(u)\not\subset proj_d(s)$. A rectangle s is stored in a set $S(u)$ for an internal node u if $rng(u)\not\subset proj_d(s)$ and $rng(u_i)\subset proj_d(s)$ for at least one child u_i of u. $S_{ij}(u)$ is the set of all intervals $s\in S(u)$ such that $rng(u_f)\subset proj_d(s)$ for a child u_f of u if and only if $i\leq f\leq j$. We store a data structure $\mathcal{D}(u)$ for $(d-1,g+1)$-stabbing queries in each internal node u. For a rectangle $s\in S_{lr}(u)$, $\mathcal{D}(u)$ contains a rectangle s' such that $proj_g(s)=proj_g(s')$ for $g\neq d$ and $proj_d(s')=[l,r]$. In other words, we replace the d-th coordinates of s's endpoints with l and r.

A query q is answered by traversing the search path for q. For a leaf node v_0, we examine all rectangles in $S(v_0)$ and find the highest priority rectangle in $S(v_0)$. In an internal node v_i, $i\geq 1$, we answer the query $q(v_i)=(q_1,\ldots,q_{d-1},f,q_{d+1},\ldots,q_{d+g})$ using the data structure $\mathcal{D}(v_i)$. The query $q(v_i)$ is obtained from q by replacing the d-th coordinate with an index f, such that v_{i-1} is the f-th child of v_i. When a node v_i is visited, our procedure finds the highest priority rectangle $s(v_i)$ that is stored in $S(v_i)$ and is stabbed by q. All rectangles that are stabbed by q are stored in some $S(v_i)$. Hence, the highest priority rectangle stabbed by q is the maximum rectangle among all $s(v_i)$. Since our procedure spends $O((\log n/\log\log n)^{d-1})$ time in each node, the total query time is $O((\log n/\log\log n)^{d})$.

When a rectangle s is inserted or deleted, we update $O(\log n/\log\log n)$ data structures $\mathcal{D}(u)$ in which s is stored. Hence, deletions and insertions are supported in $O((\log n/\log\log n)^{d})$ and $((\log n/\log\log n)^{d}\log\log n)$ time respectively. We can re-balance the base tree using a procedure that is similar to the procedure described in section 4.

Proof of Theorem 2. Theorem 2 follows easily from Lemmas 1 and 2. By Lemma 1, there exists a linear space data structure that answers $(1,d-1)$-stabbing queries in $O(\log n/\log\log n)$ time. We obtain the main result for d-dimensional stabbing-max queries by applying the result of Lemma 2 to the data structure for $(1,d-1)$-stabbing queries.

6 Stabbing-Sum Queries

We can also modify our data structure so that it supports stabbing-sum queries: count the number of intervals stabbed by a query point q.

Theorem 3 *There exists a linear space data structure that answers orthogonal stabbing-sum queries in $O(\log n/\log\log n)$ time. This data structure supports insertions and deletions in $O(\log n)$ and $O(\log n/\log\log n)$ amortized time respectively.*

The only difference with the proof of Theorem 1 is that we store data structures for counting intervals instead of $M(u)$. We maintain a data structure $X(u)$ in each internal node u. For a query

index f, $X(u)$ reports in $O(1)$ time the total number of intervals in $\cup_{l\leq f\leq r} S_{lr}(u)$. In every leaf node u_l, we maintain a data structure $Z(u_l)$ that supports stabbing sum queries on $S(u_l)$ and updates in $O(\log\log n)$ time.

As above, let $\pi = v_0 \dots v_R$ denote the search path for a query point q. In every node $v_i \in \pi$, $i \geq 1$, we count the number $n(v_i)$ of intervals in $\cup_{l\leq f\leq r} S_{lr}(u)$ using $X(v_i)$; the index f is chosen so that v_{i-1} is the f-th child of v_i. We also compute the number $n(v_0)$ of intervals that belong to $S(v_0)$ and are stabbed by q using $Z(v_0)$. An interval s is stabbed by q either if s is stored in $S(v_i)$, $i \geq 1$, and $rng(v_{i-1}) \subset s$ or if s is stored in $S(v_0)$ and s is stabbed by q. Hence, q stabs $\sum_{v_i\in\pi} n(v_i)$ intervals. Each $n(v_i)$, $i \geq 1$, can be found in $O(1)$ time and $n(v_0)$ can be found in $O(\log\log n)$ time. Hence, a query can be answered in $O(\log n/\log\log n)$ time.

Now we turn to the description of the data structure $X(u)$. Following the idea of [17], we store information about the recent updates in a word B; the array A reflects the state of the structure before recent updates. A is a static array that is re-built after $\log^{2\varepsilon} n$ updates of $X(u)$. When we construct A, we set $A[f] = \sum_{l\leq f\leq r} |S_{lr}(u)|$. The word B contains one integer value $m(l,r)$ for each pair $l \leq r$ ($m(l,r)$ can also be negative, but the absolute value of each $m(l,r)$ is bounded by $O(\log^{\varepsilon} n)$). When a segment is inserted into (deleted from) $S_{lr}(u)$, we increment (decrement) the value of $m(l,r)$ by 1. We can find $\sum_{l\leq f\leq r} m(l,r)$ in $O(1)$ time using a look-up table. After $\log^{\varepsilon} n$ updates we rebuild the array A and set all $m(l,r) = 0$. The amortized cost of rebuilding A is $O(1)$.

The total number of segments in $\cup_{l\leq f\leq r} S_{lr}(u)$ equals to $A[f] + \sum_{l\leq f\leq r} m(l,r)$. Hence, a query to $X(u)$ is answered in $O(1)$ time. We implement A in such way that each entry $A[i]$ uses $O(\log|S(u)|)$ bits. Thus each data structure $X(u)$ uses $O(\log^{\varepsilon} n \log|S(u)|)$ bits and all $X(u)$, $u \in T$, use $O(n\log n)$ bits in total.

It remains to describe the data structure $Z(u)$. Let $E(u)$ be the set that contains endpoints of all intervals in $S(u)$. Since u is a leaf node, $S(u)$ contains $O(\log^{2\varepsilon} n)$ elements. Hence, it takes $O(\log\log n)$ time to find the rank $r(q)$ of q in $E(u)$. For each $1 \leq i \leq |E|$, $C[i]$ equals to the number of intervals that are stabbed by a point q with rank $r(q) = i$. The array C enables us to count intervals stabbed by q in $O(1)$ time if the rank of q in $E(u)$ is known. Since $|E| = O(\log^{\varepsilon} n)$ and $C[i] = O(\log^{\varepsilon} n)$, the array C uses $O(\log^{\varepsilon} n \log\log n)$ bits of memory. When a set $S(u)$ is updated, we can update C in $O(1)$ time using a look-up table. Thus $Z(u)$ uses linear space and supports both queries and updates in $O(\log\log n)$ time.

References

[1] P. K. Agarwal, L. Arge, J. Yang, and K. Yi, *I/O Efficient Structures for Orthogonal Range Max and Stabbing Max Queries*, Proc. ESA 2003, 7-18.

[2] P. K. Agarwal, L. Arge, K. Yi, *An Optimal Dynamic Interval Stabbing-Max Data Structure?*, Proc. SODA 2005, 803-812.

[3] S. Alstrup, T. Husfeldt, T Rauhe, *Marked Ancestor Problems*, Proc. FOCS 1998, 534-544.

[4] L. Arge, J. S. Vitter, *Optimal External Memory Interval Management*, SIAM J. Comput. 32(6), 1488-1508 (2003).

[5] G. E. Blelloch, *Space-Efficient Dynamic Orthogonal Point Location, Segment Intersection, and Range Reporting*, Proc. SODA 2008, 894-903.

[6] B. Chazelle, *A Functional Approach to Data Structures and its Use in Multidimensional Searching*, SIAM J. on Computing, 17, 427-462 (1988).

[7] H. Edelsbrunner, *A New Approach to Rectangle Intersections*, part I, Int. J. Computer Mathematics, 13, 209-219 (1983).

[8] P. van Emde Boas, R. Kaas, E. Zijlstra, *Design and Implementation of an Efficient Priority Queue*, Mathematical Systems Theory 10, 99-127 (1977).

[9] P. Gupta and N. McKeown. *Dynamic Algorithms with Worst-Case Performance for Packet Classification*, Proc. IFIP Networking 2000, 528- 539.

[10] P. Gupta and N. McKeown. *Algorithms for Packet Classification*, IEEE Network, 15(2), 24-32 (2001).

[11] A. Feldmann, S. Muthukrishnan, *Tradeoffs for Packet Classification*, Proc. INFOCOM 2000, 1193-1202.

[12] M. L. Fredman, D. E. Willard, *Trans-Dichotomous Algorithms for Minimum Spanning Trees and Shortest Paths*, J. Comput. Syst. Sci. 48(3), 533-551 (1994).

[13] H. Imai and T. Asano, *Dynamic Orthogonal Segment Intersection Search*, Journal of Algorithms 8(1), 1-18 (1987).

[14] A. Itai, A. G. Konheim, M. Rodeh, *A Sparse Table Implementation of Priority Queues*, Proc. 8th ICALP 1981,417-431.

[15] H. Kaplan, E. Molad, R. E. Tarjan, *Dynamic Rectangular Intersection with Priorities*, Proc. STOC 2003, 639-648.

[16] Y. Nekrich, *Orthogonal Range Searching in Linear and Almost-Linear Space*, Comput. Geom. 42(4), 342-351 (2009).

[17] M. Pătraşcu, E. D. Demaine, *Tight Bounds for the Partial-Sums Problem*, Proc. 15th SODA 2004, 20-29.

[18] S. Sahni and K. Kim, $O(\log n)$ *Dynamic Packet Routing*, Proc. IEEE Symposium on Computers and Communications 2002, 443-448.

[19] S. Sahni, K. Kim, and H. Lu. *Data Structures for One-Dimensional Packet Classification Using Most Specific Rule Matching*, Proc. ISPAN 2002, 3-14.

[20] R. E. Tarjan, *A Class of Algorithms which Require Nonlinear Time to Maintain Disjoint Sets*, J. Comput. Syst. Sci. 18(2), 110-127 (1979).

[21] M. Thorup, *Undirected Single-Source Shortest Paths with Positive Integer Weights in Linear Time*, J. ACM 46(3), 362-394 (1999).

[22] M. Thorup, *Space Efficient Dynamic Stabbing with Fast Queries*, Proc. STOC 2003, 649-658.

[23] D. E. Willard, *A Density Control Algorithm for Doing Insertions and Deletions in a Sequentially Ordered File in Good Worst-Case Time*, Information and Computation 97, 150-204 (1992).

[24] J. Yang, J. Widom, *Incremental Computation and Maintenance of Temporal Aggregates*, Proc. IEEE Intl. Conf. on Data Engineering 2001, 51-60.

Appendix A. Auxiliary Data Structures

Using bitwise operations and table look-ups, we can implement the data structure $M(u)$ and the block data structures so that queries are supported in constant time.

Data Structure $M(u)$. Let $\mathcal{M}(u)$ be the set of all interval priorities stored in $M(u)$. Recall that $\mathcal{M}(u)$ contains an element $\max_{ij}$ for each set $S_{ij}(u)$, where $\max_{ij}$ is the highest priority interval (or its block label) stored in $S_{ij}(u)$. For simplicity, we assume that $\max_{ij} = -\infty$ if $S_{ij} = \emptyset$. Let the rank of x in $\mathcal{M}(u)$ be defined as $\texttt{rank}(x, \mathcal{M}(u)) = |\{\, e \in \mathcal{M}(u) \mid e \le x \,\}|$. Let $\mathcal{M}'(u)$ be the set in which every element of $\mathcal{M}(u)$ is replaced with its rank in $\mathcal{M}(u)$. That is, for each $\max_{ij}$ stored in $\mathcal{M}(u)$ we store $\max'_{ij} = \texttt{rank}(\max_{ij}, \mathcal{M}(u))$ in the set $\mathcal{M}'(u)$.

Each of $O(\log^{1/4} n)$ elements in $\mathcal{M}'(u)$ is bounded by $O(\log^{1/4} n)$. Hence, we can store $\mathcal{M}'(u)$ in one bit sequence W; W consists of $O(\log^{1/4} n \log\log n)$ bits and fits into a machine word. We can store a table $\texttt{Tbl}_1$ with an entry for each possible value of W and for each f. The entry $\texttt{Tbl}[W][f]$ contains the pair $<l, r>$, such that $\max'_{lr}$ is the highest value in the set $\{\, \max'_{ij} \mid i \le f \le j \,\}$.

Obviously, $\max_{ab} < \max_{cd}$ if and only if $\max'_{ab} < \max'_{cd}$. Therefore a query f can be answered by looking up the value $<l, r>= \texttt{Tbl}[W][f]$ for $W = \mathcal{M}'(u)$ and returning the element $\max_{lr}$.

When $M(u)$ is updated, the value of some $\max_{ij}$ is changed. We store all elements of $\mathcal{M}(u)$ in an atomic heap [12, 21] that supports predecessor queries and updates in $O(1)$ time. Hence, the new rank of $\max_{ij}$ in $\mathcal{M}(u)$ can be found in $O(1)$ time. If the rank of $\max_{ij}$ has changed, the ranks of all other elements in $\mathcal{M}(u)$ also change. Fortunately, $\max'_{ij}$ can assume only $O(\log^{1/4} n)$ values. There are $O(\log^{1/4} n)$ elements $\max'_{ij}$ in a set $\mathcal{M}'(u)$, and there are $2^{O(\log^{1/4} n)} = o(n)$ different sets $\mathcal{M}'(u)$. Besides that, each $W = \mathcal{M}'(u)$ fits into one word. Hence, we can update the set $\mathcal{M}'(u)$ using a look-up in a table $\texttt{Tbl}_1$. Suppose that the rank of the f-th element in a set $\mathcal{M}'(u)$ is changed to r. Then the new set $\mathcal{M}'(u)$ is stored in an entry $\texttt{Tbl}_1[W][f][r]$ of $\texttt{Tbl}_1$; here W is the bit sequence that corresponds to the old set $\mathcal{M}'(u)$. Thus an update of $M(u)$ can be implemented in $O(1)$ time.

We need only one instance of tables $\texttt{Tbl}$ and $\texttt{Tbl}_1$ for all nodes u of the base tree. Both tables use $o(n)$ space and can be initialized in $o(n)$ time.

Data Structure $F(B)$. The data structure $F(B)$ for a block $B \in \texttt{Comp}(u)$ is implemented as a B-tree $\mathbb{T}_F$. Every leaf of $\mathbb{T}_F$ contains $\Theta(\rho)$ identifiers and each internal node has degree $\Theta(\rho)$ for $\rho = \sqrt{\log n}$. Recall that each identifier consists of at most two indices i_1, i_2 such that the corresponding interval is stored in the i_1-th and i_2-th children of the node u in the base tree. All identifiers stored in a leaf node can be packed into $O(\rho \log\log n)$ bits. In every internal node ν of $\mathbb{T}_F$, we store the bit sequence $W(\nu)$. For every child ν_i and for every possible value of j, $W(\nu)$ contains the total number of indices j in the i-th child ν_i of ν; $W(\nu)$ consists of $O(\rho \log\log n)$ bits. Using a look-up table, we can count for any $W(\nu)$ and for any i the total number of elements stored in the children $\nu_1, \ldots, \nu_{i-1}$ of ν. For any $W(\nu)$ and i, we can also count the total number of indices j in the children $\nu_1, \ldots, \nu_{i-1}$ of ν. For any $k \le 2\log^3 n$ and any $W(\nu)$, j, we can find the largest i such that the total number of indices j in $\nu_1 \ldots, \nu_i$ does not exceed k. Look-up tables that support such queries use $o(n)$ space and can be initialized in $o(n)$ time. We need only one instance of each table for all blocks B and all $F(B)$.

All queries and updates of a data structure $F(B)$ can be processed by traversing a path in $\mathbb{T}_F$. Using the above described look-up-tables, we spend only constant time in each node of $\mathbb{T}_F$; hence, queries and updates are supported in $O(1)$ time. Details will be given in the full version of this paper. Data structures $G(B)$ and $H(B)$ can be implemented in a similar way.